Cammot-Boids Prroject Report

CPSC 565 University of Calgary

Camilo Talero
Scott Saunders

Abstract

We implemented a semi-interactive C++ program that displays multiple flocks of simple agents (called Boids in honour of the original paper by Craig Reynolds). The program uses user-editable text files to create different groups of Boids with different behaviours, and spawns obstacles, represented as spheres, that the Boids must avoid. Boids can also be controlled by the user through the mouse by clicking and positioning the cursor on the desired destination.

Boids are governed by the 3 rules originally described by Reynolds (cohesion, separation and alignment). Cohesion is the desire of each Boid to go to the average position of it's neighbours in a certain range; separation is the desire of the agents to prevent collisions by moving away from neighbours that are very close to them; alignment is the desire to match the general heading of a set of neighbours on a specific range. Certain modifications were also added, such as the addition of a field of view. Each rule is independent and each can be configured in real time through a text file.

Boids are a good example of emergent systems, since each agent is very simplistic in nature, yet complex behaviours can be achieved through modifications of a small set of parameters. They are the basis for many swarm simulations, such as the award winning film Batman Begins, which was acclaimed because of it's use of computer swarm simulations to create groups of bats.

1 Overview

1.1 Basic Rules

1.1.1 Separation

The separation rule governs the desire to avoid collisions of the Boids. There are 2 parameters that directly affect this rule.

The **sepRadius** parameter governs the radius of a sphere placed around the center of mass of the Boid that is used to determine which Boids affect the current Boid. The **sepCoeff** parameter governs the strength with which Boids reject each other. All Boids outside of this sphere will be ignored. The current Boid will calculate the average position of all Boids inside of it's separation sphere of radius **sepRadius**, and will attempt to move away from this position with an acceleration magnitude of **sepCoeff**.

1.1.2 Alignment

The alignment rule governs the desire of a Boid to match the general heading of it's neighbours. There are 2 parameters that directly influence this rule.

The **alignmentRadius** parameter determines the radius of a sphere around the center of mass of the Boid that is used to determine which Boids affect the current Boid. The **alignmentCoeff**

determines the strength of the force with which Boids accelerate to match the general heading of it's neighbours. The current Boid will ignore all Boids outside of this sphere, it will then calculate the average heading of all Boids that are inside of it's alignment sphere but outside of it's separation sphere (which forces the alignment sphere to always be bigger than the separation sphere if this rule is to be applied) and then it will accelerate so that it's heading matches that of it's neighbours with a magnitude of **alignmentCoff**.

1.1.3 Cohesion

The cohesion rule governs the desire of a Boid to head towards the general position of it's neighbours. There are 2 parameters that directly influence this rule.

The **cohesionRadius** determines the radius of the sphere around the center of mass of the Boid that is used to determine which Boids affect the current Boid. The **cohesionCoeff** parameter determines the magnitude of the force with which the current Boid accelerates towards the average position of it's neighbours inside of the cohesion sphere. All Boids outside the cohesion sphere as well as those inside the alignment or separation spheres will be ignored (forcing the radius of this sphere to be larger than that of the previous to for this rule to take effect).

1.1.4 Field of view

There is an additional rule that determines whether a Boid can see another Boid or not. If the current Boid cannot see a Boid, even if this Boid is inside any of the rule spheres (cohesion, alignment and separation) it will be ignored. There is only one parameter affecting this rule.

FOV determines the field of view of all Boids in a flock. The simplest way to abstract this rule is to imagine that inside each of the rule spheres described above, there is a conic hole of angle **FOV**, any Boid that lies inside of this conic section will be ignored by the current Boid regardless of it's position.

1.2 Additional rules

1.2.1 Herding (attractive and repelling)

This is an additional rule that allows the user to herd a flock towards/away to/from a point through the use of the mouse. There are no user configurable parameters associated with this rule.

If the user presses the left mouse button, the average position of the flock is approximated by sampling 20 Boids from the current flock. The current position of the mouse is then projected onto the 3D world with the same depth as this approximated position. All Boids in the current flock then accelerate towards this projected point, until the user let's go of the button.

If the user presses the righ mouse button, similar calculations are performed, except the Boid gets repelled from the projected point.

1.2.2 Keep Territory

This rule creates a territory for the flocks (same for all flocks, but could be implemented so that it is flock specific int he future). This creates a desire for all flocks to avoid going too far away from the starting position. There is 1 parameter associated with this rule.

radius determines the radius of a territory sphere centered at the origin $O = (0, 0, 0)$. If a Boid wanders outside this territory (i.e the length of it's position radius exceeds radius), It will ignore all other rules and will accelerate towards the origin. This essentially ensures that neither Boids nor flocks stray into infinity.

1.2.3 Collision Avoidance

This rule ensures Boids won't collide with the spherical obstacles added to the world. There are no user-editable parameters associated with this rule.

If a Boid passes a thresh hold distance (not user configurable) it will get repelled away from the center of the collision sphere with a magnitude of $k \cdot \frac{1}{distance^2}$ for a constant, implementation specific k, where distance refers to the distance form the Boid to the surface of the collision sphere. This ensures that the closer the Boid is to a collision sphere, the stronger it will be repelled away from it, and as such this force eventually overcomes the forces of any other rule if the Boid gets too close to the sphere.

1.3 Additional parameters

- **BoidNum**: the number of Boids of the flock.
- **color**: the color of the Boids in the flock (purely for aesthetical purposes).
- **center**: not fully implemented, no behaviour. Once implemented, determines the center of the territory of the flock.
- **maxAcceleration**: this parameter prevents the Boids from changing their velocity too fast. The current magnitude of the acceleration experienced by a Boid may never exceed **maxAcceleration**. If it does the net acceleration's magnitude gets changed accordingly so that it becomes **maxAcceleration**.
- **maxVelocity**: this parameter prevents the Boids from reaching velocities that are too fast. Same as with acceleration, if the velocity vector's magnitude of a Boid would exceed this value, it gets changed accordingly so as to not exceed it.

2 Implementation

The entire project was coded using C++ and OpenGL to render the objects. The additional use of the GLM and GLEW libraries was also used, both for rendering and simulation purposes. There

will be no discussion about the implementation of the basic rules described on the Overview, as this has been widely discussed in other places. For an example of a possible pseudo-code of each rule please refer to: http://www.kfish.org/Boids/pseudocode.html.

We profited from the multi-paradigmic nature of C++ and used booth it's procedural programming and object orientation aspects to structure the simulation. In particular, the simulation itself is run as a procedural program, and there is no object orientation neither on the main cpp file nor on the main loop of the program. However flocks, Boids, and collision objects were all represented as cpp objects with their respective header files. And all behaviour of their behavoiur was encoded as member functions of their class representations (e.g the behaviour rules are functions inside the Boid class).

2.1 Optimization Techniques

The most important optimization technique implemented was a technique to voxelize the space using a hash table. A cube of side length **radius** (a cube capable of containing the territory sphere) is created. This space is then voxelized and all Boids are mapped to the voxel they lie on. Each voxel is then hashed using the the i, j, k values of it's front, lower, left corner.

The following code snippets show the functions directly involved with the space hashing. The hash table is represented as a vector of vectors of Boid pointers, since each bucket in the hasTable msut contain all Boids inside the voxel mapped to that position on the hash table. Each flock has it's own hash table, included in it's class definition.

```
//hashFunction
void Flock::hash(Boid* b)
{
    if(hashTable.size() == 0)
        hashTable.resize(Boids.size()*2);

    uint hashIndex = b->x*LP1+ b->y*LP2 + b->z*LP3;
    hashIndex = hashIndex%hashTable.size();
    uint originalHI = hashIndex;

    while(hashTable[hashIndex].size()>0 &&
    (hashTable[hashIndex][0]->x!=b->x ||
    hashTable[hashIndex][0]->y!=b->y ||
    hashTable[hashIndex][0]->z!=b->z))
    {
        hashIndex = (hashIndex+1)%(hashTable.size());
        if(hashIndex == originalHI)
            return;
    }

    hashTable[hashIndex].push_back(b);
}
```

```
//voxelize the space and the Boids
void Flock::allocate()
{
    for(Boid *b: Boids)
    {
        int x = (b->position.x+radius)/cohesionRadius;
        int y = (b->position.y+radius)/cohesionRadius;
        int z = (b->position.z+radius)/cohesionRadius;

        b->x=x;
        b->y=y;
        b->z=z;

        hash(b);
    }
}
```

Although the worst case running time of the previous algorithm is $O(n^2)$, the same running time as no optimization. The use of the hash table makes it so that, as long as the Boids are not too close to each other, the worst case running time is $O(n)$ (where n is the number of Boids).

Two further optimization techniques were implemented.

The first, was to parallelize the simulation using linux pthreads when updating the Boids. The number of concurrent threads is guaranteed to be the maximum number of concurrent threads that can run simultaneously on the current CPU (machine independent). Instead of updating each Boid on one thread, the Boids get separated into *number of concurrent threads* equal sized partitions and each thread updates only those Boids associated to it. So all threads may read the information of any Boid, but can only write to those Boids associated to it, thus ensuring not 2 threads overwrite the same location in memory.

The second was to increment CPU time by requesting a higher priority to the operating system. If the prgram is run with administration priviledges (e,g run with the use of sudo in Ubuntu) it will request the operating system to give it the maximum priority (-20). Depending on the CPU load this may have little to no impact on the system however. The priority with which the program is running is displayed on the console at the beginning of the execution.

This optimization techniques allowed us to have up to 10 000 Boids in 1 flock, with acceptable lag. It is however recommended to keep the number of Boids at around 5000 to prevent unexpected lag when a large numebr of Boids clusters together.

2.2 Mathematical Model

2.2.1 Physics and Numerical Integration

In order to have a smooth and realistic looking simulation, our approach to simulate each Boid are based on physics. The abstraction is to consider each Boid a particle that can be affected by certain physical forces. We assumed all Boids had the same mass thus we simplified Newton's second law of motion $F = ma$ to simply $F = a$ or in other words, Boids accelerate based exclusively on the forces being applied to them. Each of the rules discussed in the overview section generates a vector, representing a force that is to be applied to the current Boid.

At a given time t this yields $a_t = \sum a_r$ where a_t is the acceleration at time t and $\{a_r\}$ is the set of all accelerations calculated by the behaviour rules.

This approach has the advantage that any arbitrary amount of rules can be added in a very simple manner. However it may be necessary to increase **maxAcceleration** and **maxVelocity** when doing so.

Finally, we update all parameters defined in the Boid class, following the Verlet integration approach. In other words we follow the formula:

$$X_{t+\Delta t} = X_t + (X_t - X_{t_{past}}) \cdot (\frac{\Delta t}{\Delta t_{past}}) + a_t \cdot (\frac{\Delta t + \Delta t_{past}}{2}) \cdot \Delta t$$

Where the *past* subscript denotes the value of that variable on the previous time step $t_{past} = t - \Delta t$, and X_t denotes the position of the Boid at time t.

Similarily, we update the velocity of the Boid to be:

$$v_{t+\Delta t} = \frac{X_{t+\Delta t} - X_{t_{past}}}{2\Delta t}$$

The choice of this specific numerical integration approach was not arbitrary. This Verlet integration approach does not require a constant time step, making it very suitable for real-time simulations, since, ideally, if the CPU lags, the simulation does not lag along with it, but rather it gets updated based on the real time value of t, calculated since the beginning of the execution of the program. The downside is that, due to the nature of numerical integration, a significant lag in the computer may lead to big errors in calculations.

2.2.2 Error Propagation Minimization

The reader may have noted that updating each Boid one after the other creates an error in the simulation, that propagates and becomes larger which each new updated void. Simply, if on a given time step t a Boid's position is changed, then all Boids that depended on this Boid are affected as well (e.g imagine a Boid moves away form the visual range of one of it's neighbours before they neighbour is updated, then the neighbour will not behave appropriately based on the state of the world on the current time step). To avoid this the above calculations are performed for each Boid and the results are stored. Each Boid is then updated after all calculations are performed. This prevents the error propagation described above.

2.2.3 Boid Orientation

Since this simulation focuses on visual aspects, it is important that the Boids behave in a seemingly natural way. To ensure that each Boid faces in teh direction of it's velocity, the approach is quite simple. We take the current heading of the Boid and it's current velocity and normalize both. Then we cross product the 2 to obtain the normal to the plain containing the 2 vectors, and we finally rotate the geometry of the Boid, around this normal by the angle in between the 2 vectors. More formally:

$$\hat{n} = \vec{d} \times \vec{v}$$
$$\theta = acos(\vec{d} \cdot \vec{v})$$

Once the above values are obtained we construct the corresponding rotation matrix and rotate each point in the Boid geometry by it.

3 Results

The purpose of this simulation was to create flocks of simple agents that behaved in ways that look realistic and interesting. It is not a scientific simulation of real animals, as such, the success of the project is relative to the apparent complexity of the flocks, rather than to any numerical value, statistical data or real life comparison with real animals.

Based on the above, the final result consists of a group of flocks, each composed of 1000 Boids with the default parameters and a set of collision spheres spread around randomly in the world. Each Boid has a color in order to identify which flock it belongs to, so as to make it easier to observe any emergence in the simulation. This allows us to see the first important, visual, result of the project.

At the beginning of the simulation, all Boids are evenly distributed across the space. There are no clusters nor groups of Boids on the very first moments of the simulation. However, we can see that, as time progresses, Boids of matching color (i.e Boids belonging to the same flock) Have a tendency of grouping together, and we can easily see clear, distinct Boid congregations appear over time.

Moreover, it is evident to the viewer, that, given the parameters provided with the original simulation (the original text files of the simulation before any modification by the user) Each color of Boid (each flock) has it's own unique behaviour, even if some are similar. On this set of parameters (available on a Github repository of the project, provided at the end of this document) we can describe the following behaviours per color.

- Orange: creates groups of Boids that move in the same general direction. Individual Boids exhibit a an occasional "twitch".
- Green: similar to orange, but no "twitch".
- Light Cyan: "Exploding" behaviour. Boids converge to the same position and then proceed to get away from it, and then they converge to the central position again. The Boid distribution itself inside a group is mediumly sparsed.
- Dark Blue: similar to white sometimes. However it can also create cycles (Boids orbit around a certain point) and lines (Boids follow a "leader" in a straight line).
- Magenta: has a tendency of staying on the same area. Each Boid seems to follow the same heading in a very cohesive manner, but the group as a whole is prone to turning sharply.

Based on the number of Boids achieved and the different emergent behaviours that depend solely on the described set of parameters, the simulation seems satisfactory.

Future Modifications

2 main goals were not implemented due to time constraints. The first was to add a large predator with a different set of rules, in order to display a predator prey relationship. The second was

to make the Boids have a realistic mesh (bird, fish, dragon...) and use B-Spline interpolation to animate their movement in a natural looking fashion (this is not related to emergent behaviour, adn, as such, was left as a last priority).

References

[1] Przemyslaw Prusinkiewicz, lecture slides for CPSC 587 winter 2017, University of Calgary.

[2] Craig Reynolds, http://www.red3d.com/cwr/boids/

[3] Conrad Parker, http://www.kfish.org/boids/pseudocode.html

[4] Wikipedia, https://en.wikipedia.org/wiki/Verlet_integration

[5] Open.gl, https://open.gl/

Resources

[1] C++ (GNU g++ compiler)

[2] OpenGL, https://www.opengl.org/

[3] GLEW, http://glew.sourceforge.net/

[4] GLFW, http://www.glfw.org/

[5] GLM, http://glm.g-truc.net/0.9.8/index.html

Source Code

https://github.com/ttoocs/emergent2